Colin Pham

Professor Brooks

Foundations of AI

23 September 2024

Assignment 2: Search

Question 1:

Search Algorithms

Breadth First Search

- **Time Complexity:** O(|V + E|), where V represents the number of nodes on the graph and E represents the number of edges.
- **Space Complexity:** O(|V + E|), where V represents the number of nodes present on the graph and E represents the edges. Though the graph can go over different nodes more than once, the fact still remains that there are V nodes and E edges actually present on the graph.
- **Complete?** : Yes, since the search starts from a node and expands out until it finds the solution. If a solution is accessible, then it will eventually be queued.
- **Optimal?** : Breadth First Search is not that optimal. There are cases where the solution is at the bottom of the graph and the queue takes a while to get to said solution.

Uniform Cost Search

- **Time Complexity:** O(m ^ (1+floor(L/e))) where m represents the number of neighbors a node has, L represents the shortest path, and e represents the shortest edge. The time

complexity is a little more complicated since nodes and edges can be visited more than once.

- **Space Complexity:** O(|V + E|), where V represents the number of nodes present on the graph and E represents the edges. Though the graph can go over different nodes more than once, the fact still remains that there are V nodes and E edges actually present on the graph.
- **Complete?** : Yes, uniform cost search is complete. It will go over nodes multiple times and try multiple combinations of paths until it finds the shortest and lowest cost path possible.
- **Optimal?** : No, since the search algorithm has to try every single combination of paths, which can take a while if the space is extremely large.

Depth First Search

- **Time Complexity:** O(|V + E|), where V represents the number of nodes on the graph and E represents the number of edges.
- **Space Complexity:** O(|V + E|), where V represents the number of nodes present on the graph and E represents the edges. Though the graph can go over different nodes more than once, the fact still remains that there are V nodes and E edges actually present on the graph.
- **Complete?** : No, since depth first search might go down an infinite depth that gets further and further away from the goal without being able to backtrack. The max recursion depth limit might be reached because of this.
- **Optimal?** : Not optimal since the search is able to revisit nodes that have already been visited. This is especially true in a graph where nodes are connected by more than one

connection. Also, there are cases where the solution lies in a node in a depth that is nowhere near the depths that are looked into first.

Depth Limited Search

- **Time Complexity:** O(|V + E|, level < depth limit). Basically the same as a depth first search, but the complexity is only limited to the amount of nodes and edges within the depth limit.
- **Space Complexity:** O(|V + E|, level < depth limit). Basically the same as a depth first search, but the complexity is only limited to the amount of nodes and edges within the depth limit.
- **Complete?** : Technically no, since limiting the depth of a search would mean that the algorithm can not try every possible combination of paths to the goal. If a goal is out of the limit, then the search will fail. However, if the definition states if it can reach a goal within its constraints, then I would say yes for it being complete.
- **Optimal?** : Not optimal since the user wouldn't know what depth the best solution is located in and might prevent the AI from going deep enough to get there.

Iterative Deepening Search

- **Time Complexity:** O(|V + E| ^ 2). It's basically a depth first search but with an increasing depth limit. While the depth limit keeps increasing, the number of times the algorithm goes over nodes and edges will keep increasing the deeper the search goes. It reminds me of bubble sort where the first node is gone over n times and the last node is gone over once.

- **Space Complexity:** O(|V + E|). Even though the depth of the graph is increasing, the graph as a whole stays the same.
- **Complete?** : Yes, since the search algorithm will eventually try every possible solution until the goal is reached
- **Optimal?** : It is not that optimal, especially for solutions that exist extremely deep in the graph. With how deepening search works, the time complexity will increase exponentially.

A*

- **Time Complexity:** The time complexity is a little hard to calculate since different goals and heuristics exist. How would I calculate the worst case scenario for a knowledgeable search? If I were to guess, I would say O(|V + E|) since there are some maps where a search HAS to traverse through every single node and edge to get to the optimal solution.
- **Space Complexity:** O(|V + E|). V = Nodes. E = Edges. It doesn't matter how much the search algorithm knows. If the graph is made to be traversed over entirely no matter what, then search algorithms, including A*, will traverse the entire map.
- **Complete?** I would say A* is a complete search algorithm. Since it uses knowledge on where the solution may be, it will always get there in a finite amount of time.
- **Optimal?** : Yes, it is optimal since it uses heuristics to guide it to the end. Yes, it may deviate a little from time to time, but deviations also make it so it finds optimal solutions that may have been hidden.

Question 2 Part 6 - Subproblems:

Modify your search code so that it instead solves three subproblems: moveToSample, removeSample, and returnToCharger. You can do this by changing the start state and goal test. How does this change the number of states generated?

Running the entire search from the start gives me

28 Breadth First Search States and 41 Depth First Search States

It looks like Breadth First Search eventually queued the solution while Depth First Search looked in too many depths that the solution was nowhere in. **Depth limited search** was an interesting case. In the normal Depth First Search, it took 41 states with a depth of 11. When limited to a depth of 10, the goal was reached with only 25 states generated.

I decided to divide the problem up into smaller problems. First, for my move_to_sample goal, I had the search algorithms stop as soon as the location of the rover was at the sample site. Running it gives me

3 Breadth First Search States and 5 Depth First Search States

It looks like both searches found the solution fairly quickly since the solution is extremely close to the starting point. Depth First Search seems to have taken a little longer.

Next, I had the remove_sample goal, where it starts from the sample location and had it complete when the rover extracts the sample from the site (uses the tool). Running it gives me

6 Breadth First Search States and 6 Depth First Search States

Both searches completed the move_to_sample goal in 2 steps meaning that they both still needed some time to pick up and use the tool. Again, since the goal was so close to its starting point, both searches completed with around the same efficiency

Finally, I ran the return_to_charger goal where the search algorithms had to return to the charging station after using the tool to extract the sample. Running it gives me

17 Breadth First Search States and 15 Depth First Search States

The starting point and goal are a lot further this time. This time, both searches have to worry about going from the sample site, picking up the sample, dropping off the sample, and then moving to the battery. The goal was apparently close enough for both searches to complete around the same time. Whether or not it is holding a tool is negligible for my implementation, but changing the goal to make it so the robot can either carry a tool or sample (but not both) is possible.

In conclusion, for both searches, if I were to just divide the entire problem into smaller ones and run them individually, I would end up with less states generated. However, in the case of the Mars Rover, I would rely on breadth first searches. Having Depth First Search run problem by problem is a good way to solve its inefficiency problem seen when running the whole thing altogether

Question 3 - A* and UCS:

Run both A* and uniform cost search (i.e. using h1: h=0 for all states) on the MarsMap and count the number of states generated

I was provided a map (page 2) of mars for a rover to traverse through. The white tiles are safe to move to while the red tiles are not. Starting from location 8-8, I have to navigate the rover to location 1-1.

First, I used Uniform Cost Search. This search tries every possible combination of paths in order to find the most efficient path at the end. Since it is doing absolutely everything possible, the number of states generated totaled to 32, which equates to the total number of white tiles present.

Next, I used the a* algorithm. This search is basically Uniform Cost Search but it has knowledge. If there is some sort of knowledge on how to get to the goal, then its efficiency will be increased. The number of states generated totaled to 26, which is a massive decrease

		(8,3)	x	x	x	x	Samples (8,8)
					x		
(6,1)				x	x	x	x
(5,1)	x	x	x	x			x
			x				x
			x	x	x	x	
(2,1)						x	
Charger (1,1)	(1,2)	x	x	x	x	x	

Question 5 - Deep Blue vs Alpha Zero

Deep Blue was an artificial intelligence that was successful in beating a grandmaster level chess player at their own game. Its success mainly stems from the fact that it's a culmination of learned mistakes and improvement from previous iterations. Such improvements revolved around the chips it runs on. Because of Deep Blue's 216 chess chips, it was able to visualize 2.5 million chess moves and calculate an optimal move given a scenario. Its next iteration, Deep Blue II, featured not only twice the amount of chips, but also a way to remove duplicate moves, thus enhancing its calculation speeds. How the AI actually worked with the data was the one thing that was focused on. In the end, game specific tuning was the cause of its victory against Garry Kasparov. Other problems that require the knowledge of every possible combination can be used with the kind of chips Deep Blue utilized. Also, since it favors a tree-like structure and search, any sequence based problem (or any problem that can be looked at as one) is compatible with these chips. The real problem, the evaluation step, stems from learning from past mistakes. Thus, previous chess games can only benefit chess specifically while previous problem attempts can only benefit said problems.

Alpha Zero and Stockfish were put head to head in chess with Alpha Zero winning all of its games despite having 1/10 of Stockfish's calculation time. This is all due to its method of calculation. Though it uses Monte Carlo to play every single possible combination, it calculates the probability of a desired outcome rather than a win. Trying to calculate every single chess possibility at once is long. Dividing the problem up into smaller ones and playing to get to certain steps is more optimal and leaves room to account for deviations. To frame it in a simpler light, think about a chess game with a starting state and a winning state. A victorious chess game consists of different steps, or nodes. Say there are multiple scenarios of a chess game, and nodes

a, b, and c are the outcomes needed in order to win said scenarios. Calculating every possible scenario and steps would be cumbersome, especially during the start state where anything can happen. Instead, start from the beginning and work on node a, in which a more concise strategy can be calculated more easily depending on the opponent's move so far. Doing the same for nodes b and c would result in a faster calculation.